Learning Programming by applied activities: an example with topics of Operating Systems

Eva Gibaja[1][0000−0002−0184−8789], María Luque[1][0000−0001−7735−8340], and Amelia Zafra[1][0000−0003−3868−6143]

University of Córdoba, Department of Computing and Numerical Analysis
{egibaja, mluque, azafra}@uco.es

Resumen When a teacher is preparing a collection of exercises, a connection between the subject and the context of the degree would be desirable. Nevertheless, finding practical examples of the topics of the subject with application to other subjects is occasionally difficult. This work presents a practical proposal for simultaneously practicing concepts of computer programming and operating systems which faces the students to a real problem. In this way the students are more motivated increasing their success probabilities. Among the topics of operating systems, process scheduling has been chosen for practice is computer programming. In general terms, a scheduler manages which process will be executed in a certain moment. Some of the strategies used to perform this management employ FIFO or LIFO structures, which are typical contents of computer programming. Thereby, the development and implementation of a scheduler would allow students applying and reinforcing these concepts. The proposal may be interesting for teachers of Programming, Data Structures and Operating Systems in a Computer Science degree.

Keywords: educational resource · operating systems · programming · scheduler · files · stack · queue.

1. Introduction

When a student has to solve a problem or exercise, it is more engaging if it is related with the scope of the study program as the student sees a practical application of what it is being studied. Nevertheless, frequently occurs that finding practical and simple applied examples is not easy, even more if the subject is part of the block of foundations which are taught in the first course.

This is common in programming subjects in the degree in *Computer Engineering*, in which finding enough simple examples to practice the contents in a real problem is hard. Applying the tools and knowledge provided in programming subjects to implement concepts related to computing could be a solution. In our case, we have focused in the field of operating systems. This way, our objective is two-fold. On the one side, we intend to motivate students while on the other hand we intend to increase coordination among different subjects and courses.

Tables 1 and 2 show, respectively, a summary of the competences and topics of the subjects *Programming Methodology* (PM) and *Operating Systems* (OS) in the degree in Computer Engineering of the University of Córdoba. Particularly, in PM the C language is introduced. This language has a broad historical development with application in operating systems, compilers, software development and it is specifically oriented to the implementation of operating systems (e.g. UNIX, Windows or GPU/Linux). Besides, according to the CEB4 competence, PM should provide students with basic knowledges in operating systems.

The relationship between these two subjects is clear, and due to this reason we consider that a practical programming project in the context of operating systems is a good proposal. not only to practice programming applied to a real problem [1] [2] [3], but also to introduce some concepts related to OS [5], [6], [7], [8], [9], [10], [11], [12].

Table 1. Course Programming Methodology.

Competences
CB4: Being able to communicate information, ideas, problems and solutions to both specialized and not specialized public
CU2: Knowing and improving the user level in the context of ICTs
CEB4: Basic knowledge about programming and using computers, operating systems, databases and software related with engineering
CEB5: Knowledge about the structure, organization, operation and interconnection of computing systems. Foundations of programming and its application to solve engineering problems
Topics
Pointers
Text and binary files
Structure of a executing program
Dynamic memory
Recursivity
Dynamic linear data structures: lists, stacks and queues
Basic sorting and searching algorithms and their complexity
Methodological issues of programming: documentation and tests
Programming tools: automatic generation of projects, documentation, libraries, debuggers

2. Proposal: Programming a mini-scheduler

The subject OS covers topics such as threads, processes, scheduling or distributed systems (Table 2) which offer the opportunity to be used as object of a programming project. In this case, the processes scheduling has been used as conducting thread as it is an easy to understand concept without needing

Table 2. Course Operating Systems.

Competences
CEC10: Knowledge about features, functionalities and structure of operating systems and how to design and implement applications based on their services
CEC11: Knowledge and application of features, functionalities, structure and how to design and implement application based on distributed systems, networks and Internet
CEC14: Knowledge and application of foundations and basic techniques of parallel computing, concurrent, distributed and real time programming
Topics
Introduction to operating systems, organization, structure and operation
Processes and threads, representation, states and life cycle
Communication among processes and threads, mutual exclusion, application of synchronization to classic problems
Scheduling algorithms and their influence on the performance of the system, advantages and disadvantages
Introduction to distributed systems, common problems and basic communication paradigms
Basic concepts of input/output, memory, storing
Introduction to types of free software licenses and their relationship with GNU/Linux

a previous or deep knowledge about OS. Besides, it allows us to cover a great portion of topics in PM (Table 3).

Table 3. Topics covered by the project.

	Dynamic memory	Files	Queues	Stacks	Reference parameters	Makefiles	Arguments to main	Libraries
Processes	√	√				√	√	√
Scheduling	√		√	√	√	√	√	√

2.1. The scheduler

The scheduler is one of the main components of any modern multi-task operating system. Its main aim is assigning the CPU time to the different executing processes according to any scheduling strategy, for example (Figure 1):

- *First Come First Served* (FCFS). If a job has arrived first it is first served.
- *Shortest Job First* (SJF). The shortest job is attended first.
- *Priority-based Scheduling*. Each process is assigned a priority and the process with highest priority is executed first.

- *Round Robin* (RR). Each process is provided a fix time to execute (i.e. *quantum*). Once a job is executed for the given period, it is preempted and other process executes for a given time period.

Multi-task operating systems are characterized by being capable of developing many tasks at the same time. This feature, which is essential for users, is actually a kind of illusion produced by the scheduler as microprocessor is able to run just one single process simultaneously. Therefore, in machines with a single processor, when several processes need to be run, the scheduler distributes the CPU time assigning to processes very small-time intervals in which the process is run. As these intervals are very small (at the order of milliseconds), the user has the feeling that processes are running at the same time.

The scheduler has also another essential task for operating systems: optimizing the use of available resources (i.e. memory, hard drives, keyboard, printer and other devices). When a process requires using a certain resource, it is temporarily blocked until the resource is assigned to the process. For example, when a program is requiring the user input data, several seconds, even minutes, may pass until the user reads the requirement, thinks and types the answer and the keyboard transmits it to the program. As in all this elapsed time the program does not really need the computation time assigned, the scheduler will remove it from CPU until the execution can continue.

2.2. How the mini-scheduler works

Let's suppose a set of processes to be scheduled whose description (*name, CPU cycles needed to finish, priority*[1]) is stored into a text file. The scheduler, in each cycle, performs the following steps (Figure 2):

1. If the end of the processes file has not been reached, read the following process, P_i.
2. If CPU is busy, check priority of the process in CPU, P_{cpu}.
   - If P_{cpu} has more priority than P_i, then put P_i into the queue of processes, Q, according to its priority.
   - If P_i has more priority than P_{cpu}, then remove P_{cpu} from CPU and assign computing time to the new process P_i. When the scheduler removes a process from CPU, it does not return back to the queue of processes, but it is put into another data structure with all processes removed from CPU which has a special priority. Particularly, all these processes are introduced into a stack, S.
3. If CPU is empty, put P_i into the queue of processes Q. Then remove the first element in the stack S, or of the queue just in case S is empty and put the process in CPU.
4. When this point is reached, a CPU cycle has finished. In order to represent this fact, the number of cycles needed to finish the process at CPU is decreased 1 cycle.

[1] This priority could be established according to different criteria: the priority of the process or the number of CPU cycles remaining to end the execution.

Process	Arrival Time	Priority	CPU Cycles
P1	0	2	2
P2	1	3	2
P3	2	0	1

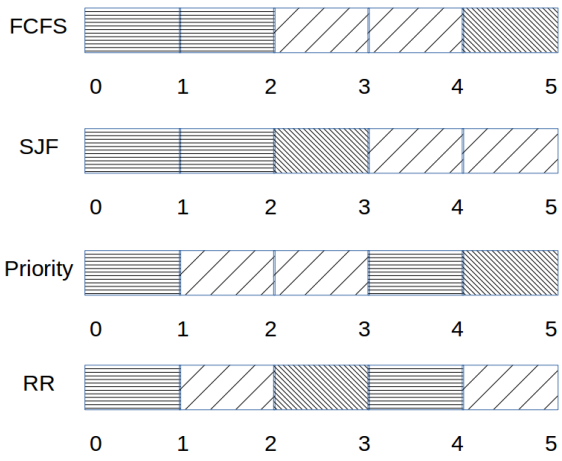

Figure 1. Scheduling examples.

5. If the number of CPU cycles needed to finish the process reaches zero, the process has finished and CPU is empty again.

The steps defined above are repeated until Q and P are empty and all the lines in the file of processes have been read.

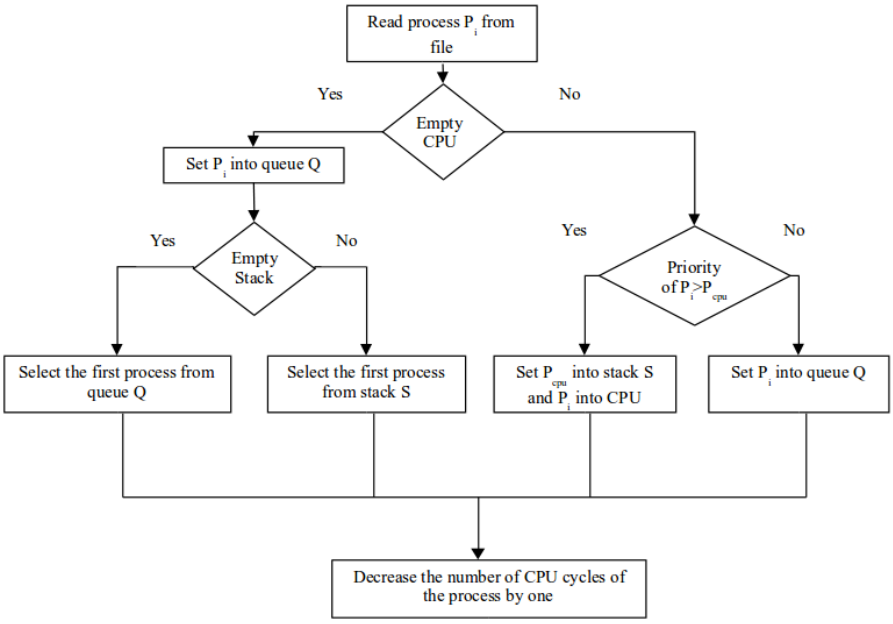

Figure 2. Scheme of the mini-scheduler.

2.3. Description of the project

In this project, the mini-scheduler described in the previous section must be programmed by using the functions the student considers appropriate[2].

The processes to be scheduled will be read from a text file. This file will have one line per process. Each of these lines will consist of three integers separated

[2] It is obvious that a real scheduler is an extremely complex element with high influence in the efficiency of the operating system. In this project, a very schematic version will be developed that will be a vehicle to practice programming skills.

by a blank. The first one represents the identifier of the process, the second one the number of CPU cycles need by the process to finish, and the last one is the priority of the process. An example is available in Table 4. For instance, if the line *3 1 0* is read, the identifier of the process is 3, it needs 1 CPU cycle to finish and has priority 0.

Table 4. Example of an input file.

1 2 2
2 2 3
3 1 0

The output of the program will be written in a text file summarizing the processes using CPU each cycle (Table 5). In each line will be printed:

- CPU cycle. An integer beginning in 1.
- Identifier of the process beginning with a *P* character.
- If the process is assigned to the CPU for the first time, this fact will be represented by >> and if the process finishes its execution it will be represented by <<.
- The number of cycles to finish.
- Finally, the priority.

Table 5. Example of an output file.

1 P1 >> 2 2
2 P2 >> 2 3
3 P2 << 1 3
4 P1 << 1 2
5 P3 >><< 1 0

The program will be executed by typing:

schedule <processesFile><outputFile><p|c>

where:

- *processesFile* is the name of the file with processes.
- *outputFile* is the name of the file with the output of the scheduler.
- *p|c* determines the order in the queue: priority (*p*) or number of cycles to finish (*c*).

Arguments to main will be used and the number and type of the arguments will be checked. In case of any error, the program will finish showing the user an error message with the right call.

To develop the mini-scheduler, a *struct process* with the following fields will be used (Figure 3):

- *pid*, an integer representing the identifier of the process. Each process will have an unique *pid*.
- *cic*, an integer representing the number of CPU cycles remaining to finish the process.
- *pri*, an integer representing the priority of the running process.

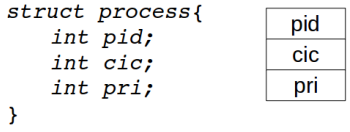

Figure 3. struct process.

By using this *struct process* the following data structure will be build (Figures 4 and 5):

- A structure to emulate the CPU that will contain an element of type *struct process*.
- A stack, *S*, with processes removed from CPU. The stack will implement, at least, the functions *push*, *pop* and *isEmpty*.
- A queue, *Q*, of processes. This queue works as a queue in which elements are sorted according to one of the following criteria:

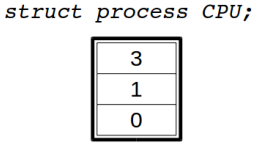

Figure 4. Emulated CPU

- *Number of CPU cycles remaining to finish*. Those processes with less cycles remaining to be completed will be selected first.
- *Priority of the process*. Processes with the highest priority will be selected first. For instance, a process with priority 10 will be served before a process with priority 3.

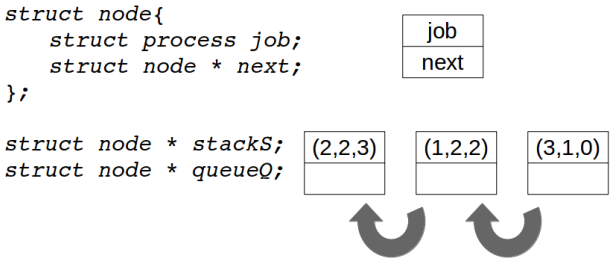

Figure 5. Emulated stack and queue

The queue will implement, at least, the functions *isEmpty*, *delete* and *insert* according to one of the criteria described above for which pointer to functions will be used. The insert function will have the following header: *void insert(struct cola ** head, int (* sort)(struct process a, struct process b), struct process p).*

The project will be organized into the following files: *queue.c*, *queue.h* (functions and *struct* needed to implement the queue), *stack.c* y *stack.h* (functions and *struct* needed to implement the stack), *files.c*, *files.h* y *main.c*. Preprocessor directives will be used to avoid including duplicated header files.

A *makefile* will be used to build the executable file. This file must have, at leas:

- Independent rules for each *.o*
- A rule to generate a library *linkedStructs.a* with the functions for the stack and the queue.
- A rule to generate the executable from *linkedStructs.a* and *main.o.*
- A rule to clean the intermediate files.
- A target to generate the executable and clean targets.
- A *phony* target.

Conclusions

This work has described an educational resource to practice topics of a programming course contextualized in the field of operating systems in a degree in Computer Sciences at the University of Córdoba. Due to the foundational character of both subjects in the study program, this proposal can also be useful as a resource for other teachers in similar subjects in other degrees and universities. The main contributions of the proposal are highlighted below:

- The fact that the student has to design and implement the mini-scheduler lead to learning by means of the development of projects. This will lead to a better comprehension of the concepts being applied and to establish relationships among them.
- The implementation of the mini-scheduler allows practicing topics of the subject PM in a real problem in the field of computing. This is motivating and engaging for all the related subjects.
- With this proposal that develops a mini-scheduler, the competence CEB4 of PM, whose aim is providing the student with foundational knowledge in operating systems, is covered.

Referencias

1. L. Joyanes, I. Zahonero. Programación en C. Metodología, algoritmos y estructuras de datos. McGraw-Hill, 2005.
2. L. Joyanes, A. Castillo, L. Sánchez, I. Zahonero. Programación en C: libro de problemas. McGraw-Hill, 2003.
3. Kernigham, N. B., Ritchie, M. D. El lenguaje de programación C. Prentice-Hall. 1989
4. W. Stallings. Sistemas operativos, 5 edición. Prentice Hall, Madrid, 2005.
5. A. S. Tanenbaum. Sistemas operativos modernos. 3 edición, Prentice Hall, Madrid, 2009.
6. A. Silberschatz, P. B. Galvin, G. Gagne. Fundamentos de Sistemas Operativos, Séptima edición. Mc Graw Hill,2005.
7. S. Candela, C. Rubén, A. Quesada, F. J. Santana, J. M. Santos. Fundamentos de Sistemas Operativos, teoría y ejercicios resueltos. Paraninfo, 2005.
8. A. Mclver, I. M. Flynn. Sistemas Operativos, Sexta edición. Cengage Learning, 2011.
9. J. Aranda, M. A. Canto, J. M. de la Cruz, S. Dormido, C. Mañoso. Sistemas Operativos: Teoría y problemas. Sanz y Torres S.L, 2002.
10. J. Carretero, F. García, P. de Miguel, F. Pérez, Sistemas Operativos: Una visión aplicada. Mc Graw Hill, 2001.
11. K. A. Robbins, S. Robbins. UNIX Programación práctica. Prentice Hall, 1997
12. K. A. Robbins, S. Robbins. Unix Systems Programming. Prentice Hall, 2003.